Non-IEEE Division, Square Root, Reciprocal, and Reciprocal Square Root Algorithms for the Intel® Itanium™ Architecture

Application Note

November 2003

Document Number: 300141-001

Contents

Revision History

Rev.	Draft/Changes	Date
—001	• Initial Release	November 2003

1. Introduction

1.1. Background

The algorithms for non-IEEE floating-point division, square root, reciprocal, and reciprocal square root presented in this document are derived from IEEE-correct floating-point division and square root algorithms complying with the IEEE Standard 754-1985 for Binary Floating-Point Arithmetic [1], and whose accuracy ensures errors of at most 0.5 ulp (units-in-the-last-place). The IEEE-correct algorithms are available online in Document Nr. 248725-004, *Division, Square Root, and Remainder Algorithms for the Intel® Itanium™ Architecture* [2] (see also [3] to [10]).

Note that the IEEE Standard does not define the reciprocal and reciprocal square root operations. However, the reciprocal operation is equivalent to a division where the numerator is 1.0, and the reciprocal square root viewed in the spirit of the IEEE Standard should return the precise value of the result, correctly rounded to the destination precision.

The operations performed by the non-IEEE algorithms are slightly less accurate, but faster than their equivalent IEEE-correct algorithms. The former might produce results with larger relative errors than the latter for certain operands, but the error is never more than 1 ulp when the rounding-to-nearest mode is used, and not more than 1.5 ulp for the other three rounding modes defined by the IEEE Standard.

The non-IEEE algorithms are not unique – a trade-off had to be made between speed and accuracy. The algorithms in this document were designed to have an error of no more than one ulp when the rounding-to-nearest mode is used. Other constraints could lead to different algorithms, with different performance and accuracy. For example, an extreme case could be to calculate the result of a division operation with just one **frcpa** and one **fmpy** instruction, but the accuracy would be very low. The algorithms presented here have better performance than the IEEE-correct ones, and are accurate enough so that they can be used as a substitute in most cases.

1.2. Performance and Accuracy of the Non-IEEE Division, Square Root, Reciprocal, and Reciprocal Square Root Algorithms

The theoretical error bounds for the non-IEEE operations are specified in the next table. These theoretical values are guaranteed upper bounds, but might not be reached in some cases. For this reason, maximum errors observed in testing are also included in the table (but any rigorous error analysis should be based on the theoretical bounds). In each case, the same algorithm can be used both as a latency-optimized, and as a throughput-optimized algorithm (for software-pipelined loops). Note that latency is defined as the number of clock cycles between starting the operation and having the result available, and throughput is defined as the number of clock cycles used to execute an operation, averaged over a

large number of independent instances. The latency and throughput of the IEEE-correct algorithms for the Itanium 2 processor are also shown for comparison purposes.

Operation	Non-IEEE Latency (clock cycles)	IEEE Latency (clock cycles)	Non-IEEE Throughput (clock cycles/ operation)	IEEE Throughput (clock cycles/ operation)	Non-IEEE Theoretical Accuracy (ulps)	Non-IEEE Observed Accuracy (ulps)	IEEE Accuracy (ulps)
Single Precision Division	16	24	2.5	3.5	0.6585	0.6524	0.50
Double Precision Division	20	28	4.0	5.0	0.5018	0.5010	0.50
Single Precision Square Root	20	28	3.0	5.0	0.9449	0.8194	0.50
Double Precision Square Root	32	36	5.5	6.5	0.5001	0.5000	0.50
Single Precision Reciprocal	16	24	2.0	3.5	0.6585*	0.6487	0.50
Double Precision Reciprocal	20	28	3.5	5.0	0.5010	0.5003	0.50
Single Precision Reciprocal Square Root	20	52 (sqrt+div)**	3.5	8.5 (sqrt+div)**	0.9449	0.8860	N/A**
Double Precision Reciprocal Square Root	32	64 (sqrt+div)**	6.5	11.5 (sqrt+div)**	0.5031	0.5007	N/A**

* An alternate algorithm having one more instruction (5 instructions instead of 4) is available for single precision reciprocal, with accuracy of 0.5004 ulp and observed maximum error of 0.5000.

** The reciprocal square root cannot be calculated easily ‘in the IEEE spirit’. For this reason, a square root operation followed by a division can be used as a comparison alternative to the non-IEEE algorithm. The resulting error can be of up to 2 ulps, hence the ‘N/A’ entries above. However, the reciprocal of a floating-point number can be calculated easily ‘in the spirit of the IEEE standard’ using one division, and the error will be at most 0.5 ulp.

1.3. Floating-Point Software Assistance

Floating-Point Software Assistance (FPSWA) requests can be issued for different reasons in the Intel® Itanium™ architecture, depending on the implementation [9], [10]. In the Itanium and Itanium 2 processor implementations, FPSWA requests for the non-IEEE

floating-point division, square root, reciprocal, and reciprocal square root algorithms from this document can occur (1) for denormal operands (FPSWA faults), and (2) when tiny results are generated (FPSWA traps). These will cause a system software component, the FPSWA handler, to compute the correct result for the excepting instruction, through software emulation.

Special cases occur for the reciprocal approximation instructions when at least one operand is zero, infinity, or NaN (or negative for **frsqrta**). In such situations, the initial approximation instructions will clear their output predicate register. As a consequence the instructions in the body of the algorithm, which are predicated on this register, will not be executed and the reciprocal approximation instructions provide the result instead. However, for the reciprocal square root operation the result from the **frsqrta** instruction is not the expected one, which is obtained further from an **frcpa** instruction applied to the result of the **frsqrta**.

For correct operation of the software sequences presented here, it is imperative to use the same destination register for the first instruction of the computation (the reciprocal approximation instruction (**frcpa** or **frsqrta**), and for the last one (an **`fma`** instruction that generates the final result). In this way, whether a FPSWA fault occurs or not, or whether **frcpa**/**frsqrta** generate directly the answer, the result of the scalar computation will be provided correctly in the destination register of the last instruction. Also, all the instructions performing intermediate calculations have to use status field `s1` from the FPSR (Floating-Point Status Register). The first and the last instruction in each sequence use `s0`, the user status field from the FPSR.

2. Non-IEEE Floating-Point Division

Two algorithms are provided: one for single precision, and one for double precision division operations.

2.1. Single Precision Floating-Point Division, Non-IEEE

This algorithm calculates $q = a/b$ in single precision, where a and b are single precision numbers. *rn* is the IEEE round-to-nearest mode, and *rnd* is any IEEE rounding mode. All other symbols used are 82-bit floating-point register format numbers. The precision used for each step is shown below. The result has an error of less than 0.6585 ulp, but most often of less than 0.5 ulp. (Algorithm derived from Doc. Nr. 248725-004, section 2.2)

Itanium 2 latency: 16 clock cycles (24 clock cycles for IEEE-correct division)

Itanium 2 throughput: 2.5 clock/operation cycles (3.5 clock cycles/operation for IEEE-correct division)

(1) $y_0 = 1 / b \cdot (1+\varepsilon_0),\ |\varepsilon_0|<2^{-8.886}$ table lookup

(2) $e_0 = (1 - b \cdot y_0)_{rn}$ 82-bit floating-point register format

(3) $q_0 = (a \cdot y_0)_{rn}$ 82-bit floating-point register format

(4) $e_1 = (e_0 + e_0 \cdot e_0)_{rn}$ 82-bit floating-point register format

(5) $q_1 = (q_0 + e_1 \cdot q_0)_{rnd}$ single precision

The assembly language implementation:

```
  .file "sgl_div.s"
  .section .text
  .proc sgl_div#
  .global sgl_div#
  .align 32

sgl_div:

{ .mii
  alloc r31=ar.pfs,3,0,0,0 // r32, r33, r34

  // &a is in r32
  // &b is in r33
  // &div is in r34 (the address of the division result)

  nop.i 0
  nop.i 0
} { .mmi
  // load a, the first argument, in f6
  ldfs f6 = [r32]
  // load b, the second argument, in f7
  ldfs f7 = [r33]
  nop.i 0;;
}

  // BEGIN NON-IEEE SINGLE PRECISION DIVISION ALGORITHM
  // general registers used: none
  // predicate registers used: p6
  // floating-point registers used: f6, f7, f8

{ .mfi
  nop.m 0
  // Step (1) y0 = 1 / b in f8
  frcpa.s0 f8,p6=f6,f7
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (2) e0 = 1 - b * y0 in f7
  (p6) fnma.s1 f7=f7,f8,f1
  nop.i 0
} { .mfi
  nop.m 0
  // Step (3) q0 = a * y0 in f8
  (p6) fma.s1 f8=f6,f8,f0
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (4) e1 = e0 + e0 * e0 in f6
  (p6) fma.s1 f6=f7,f7,f7
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (5) q1 = q0 + e1 * q0 in f8
  (p6) fma.s.s0 f8=f6,f8,f8
  nop.i 0;;
}
```

```
  // END NON-IEEE SINGLE PRECISION DIVISION ALGORITHM

{ .mib
  // store result
  stfs [r34]=f8
  nop.i 0
  // return
  br.ret.sptk b0;;
}

  .endp sgl_div
```

Sample test driver:

```
#include <stdio.h>
void sgl_div (float *, float *, float *);

void run_test (unsigned int ia, unsigned int ib) {
  float a, b, q, q0;
  unsigned int iq;

  *(unsigned int *)(&a) = ia;
  *(unsigned int *)(&b) = ib;
  q0 = a / b;
  iq = *(unsigned int *)(&q0);

  sgl_div (&a, &b, &q);

  printf ("\nNumerator: %lx\nDenominator: %lx\nQuotient: %lx\n",
      ia, ib, iq);
  if (iq == *(unsigned int *)(&q)) {
    printf ("Passed\n");
  } else {
    printf ("Failed: q = %8.8x\n", *(unsigned int *)(&q));
    exit (1);
  }
}

void main () {
  run_test (0x3f800000,0x3f800000); // 1 / 1=1
  run_test (0x3f800000,0x00000000); // 1 / 0=Infinity
  run_test (0xbf800000,0x7f800000); // -1 / Infinity=-Zero
  run_test (0x00000000,0xff800000); // +0 / -inf
  run_test (0x00000000,0x00000000); // +0 / +0
  run_test (0x80000000,0x00000000); // -0 / +0
  run_test (0x7f800000,0x80000000); // +inf / -0
  run_test (0x7f800000,0xff800000); // +inf / -inf
  run_test (0x3f800001,0x3f800002); // 1+1u / 1+2u
  run_test (0x3f8e676a,0x3f8fc87d);
  printf ("\n\tALL TESTS PASSED\n");
}
```

2.2. Double Precision Floating-Point Division, Non-IEEE

This algorithm calculates $q = a/b$ in double precision, where a and b are double precision numbers. *rn* is the IEEE round-to-nearest mode, and *rnd* is any IEEE rounding mode. All other symbols used are 82-bit floating-point register format numbers. The precision used

for each step is shown below. The result has an error of less than 0.5018 ulp, but most often of less than 0.5 ulp. (Algorithm derived from Doc. Nr. 248725-004, section 2.5)

Itanium 2 latency: 20 clock cycles (28 clock cycles for IEEE-correct division)

Itanium 2 throughput: 4 clock cycles/operation (5.0 clock cycles/operation for IEEE-correct division)

(1)	$y_0 = 1 / b \cdot (1+\varepsilon_0),\ \ \lvert\varepsilon_0\rvert<2^{-8.886}$	table lookup
(2)	$d = (1 - b \cdot y_0)_{rn}$	82-bit floating-point register format
(3)	$q_0 = (a \cdot y_0)_{rn}$	82-bit floating-point register format
(4)	$d_2 = (d \cdot d)_{rn}$	82-bit floating-point register format
(5)	$d_3 = (d \cdot d + d)_{rn}$	82-bit floating-point register format
(6)	$d_5 = (d_2 \cdot d_2 + d)_{rn}$	82-bit floating-point register format
(7)	$q_1 = (q_0 + d_3 \cdot q_0)_{rn}$	82-bit floating-point register format
(8)	$q_2 = (q_0 + d_5 \cdot q_1)_{rnd}$	double precision

The assembly language implementation:

```
  .file "dbl_div.s"
  .section .text
  .proc dbl_div#
  .global dbl_div#
  .align 32

dbl_div:

{ .mii
  alloc r31=ar.pfs,3,0,0,0 // r32, r33, r34

  // &a is in r32
  // &b is in r33
  // &div is in r34 (the address of the division result)

  nop.i 0
  nop.i 0
} { .mmi
  // load a, the first argument, in f6
  ldfd f6 = [r32]
  // load b, the second argument, in f7
  ldfd f7 = [r33]
  nop.i 0;;
}

  // BEGIN NON-IEEE DOUBLE PRECISION DIVISION ALGORITHM
  // general registers used: none
  // predicate registers used: p6
  // floating-point registers used: f6, f7, f8, f9

{ .mfi
  nop.m 0
  // Step (1) y0 = 1 / b in f8
  frcpa.s0 f8,p6=f6,f7
  nop.i 0;;
} { .mfi
  nop.m 0
```

```
  // Step (2) d = 1 - b * y0 in f7
  (p6) fnma.s1 f7=f7,f8,f1
  nop.i 0
} { .mfi
  nop.m 0
  // Step (3) q0 = a * y0 in f8
  (p6) fma.s1 f8=f6,f8,f0
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (4) d2 = d * d in f6
  (p6) fma.s1 f6=f7,f7,f0
  nop.i 0
} { .mfi
  nop.m 0
  // Step (5) d3 = d * d + d in f9
  (p6) fma.s1 f9=f7,f7,f7
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (6) d5 = d2 * d2 + d in f6
  (p6) fma.s1 f6=f6,f6,f7
  nop.i 0
} { .mfi
  nop.m 0
  // Step (7) q1 = q0 + d3 * q0 in f9
  (p6) fma.s1 f9=f9,f8,f8
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (8) q2 = q0 + d5 * q1 in f8
  (p6) fma.d.s0 f8=f9,f6,f8
  nop.i 0;;
}

  // END NON-IEEE DOUBLE PRECISION DIVISION ALGORITHM

{ .mib
  // store result
  stfd [r34]=f8
  nop.i 0
  // return
  br.ret.sptk b0;;
}

  .endp dbl_div
```

Sample test driver:

```
#include <stdio.h>
void dbl_div (double *, double *, double *);

void run_test (unsigned __int64 ia, unsigned __int64 ib) {
  double a, b, q, q0;
  unsigned __int64 iq;

  *(unsigned __int64 *)(&a) = ia;
  *(unsigned __int64 *)(&b) = ib;
  q0 = a / b;
  iq = *(unsigned __int64 *)(&q0);

  dbl_div (&a, &b, &q);

  printf ("\nNumerator: %I64x\nDenominator: %I64x\nQuotient: %I64x\n",
      ia, ib, iq);
  if (iq == *(unsigned __int64 *)(&q)) {
    printf ("Passed\n");
```

```
  } else {
    printf ("Failed: q = %I64x\n", *(unsigned __int64 *)(&q));
    exit (1);
  }
}

void main () {
  run_test (0x3ff0000000000000, 0x3ff0000000000000); // 1 / 1
  run_test (0x3ff0000000000000, 0x0000000000000000); // 1 / 0
  run_test (0xbff0000000000000, 0x7ff0000000000000); // -1 / inf
  run_test (0x0000000000000000, 0xfff0000000000000); // +0 / -inf
  run_test (0x0000000000000000, 0x0000000000000000); // +0 / +0
  run_test (0x8000000000000000, 0x0000000000000000); // -0 / +0
  run_test (0x7ff0000000000000, 0x8000000000000000); // +inf / -0
  run_test (0x7ff0000000000000, 0xfff0000000000000); // +inf / -inf
  run_test (0x3ff0000000000001, 0x3ff0000000000002); // 1+1u / 1+2u
  run_test (0x3f8e676a8878987b, 0x3acff9de398fc87d);
  printf ("\n\tALL TESTS PASSED\n");
}
```

3. Non-IEEE Floating-Point Square Root

Two algorithms are provided: one for single precision, and one for double precision square root operations.

3.1. Single Precision Floating-Point Square Root, Non-IEEE

This algorithm calculates $S = \sqrt{a}$ in single precision, where *a* is a single precision number. *rn* is the IEEE round-to-nearest mode, and *rnd* is any IEEE rounding mode. All other symbols used are 82-bit floating-point register format numbers. The precision used for each step is shown below. The result has an error of less than 0.9449 ulp, but often of less than 0.5 ulp. (Algorithm derived from Doc. Nr. 248725-004, section 3.1)

Itanium 2 latency: 20 clock cycles (28 clock cycles for IEEE square root)

Itanium 2 throughput: 3.0 clock cycles/operation (5.0 clock cycles/operation for IEEE square root)

(1)	$y_0 = \left(1 / \sqrt{a}\right) \cdot (1+\varepsilon_0),\ \ \lvert\varepsilon_0\rvert < 2^{-8.831}$	table lookup
(2)	$S_0 = (a \cdot y_0)_{rn}$	82-bit floating-point register format
(3)	$d = (1 - S_0 \cdot y_0)_{rn}$	82-bit floating-point register format
(4)	$e = (0.5 + 0.375 \cdot d)_{rn}$	82-bit floating-point register format
(5)	$T_0 = (d \cdot S_0)_{rn}$	82-bit floating-point register format
(6)	$S = (S_0 + e \cdot T_0)_{rnd}$	single precision

The assembly language implementation:

```
  .file "sgl_sqrt.s"
  .section .text
  .proc sgl_sqrt#
  .global sgl_sqrt#
  .align 32

sgl_sqrt:

{ .mmi
  alloc r31=ar.pfs,2,0,0,0  // r32, r33

  // &a is in r32
  // &sqrt is in r33 (the address of the sqrt result)

  // load the argument a in f6
  ldfs f6 = [r32]
  nop.i 0
}

  // BEGIN NON-IEEE SINGLE PRECISION SQUARE ROOT ALGORITHM
  // general registers used: r2
  // predicate registers used: p6
  // floating-point registers used: f6, f8, f7, f9, f10

{ .mlx
  nop.m 0
  // +1/2 in f9
  movl r2 = 0x3f000000;;
} { .mlx
  setf.s f9 = r2
  // +3/8 in f10
  movl r2=0x3ec00000;;
} { .mfi
  setf.s f10 = r2

  // Step (1) y0 = 1/sqrt(a) in f7
  frsqrta.s0 f7,p6=f6
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (2) S0 = a * y0 in f6
  (p6) fma.s1 f6=f6,f7,f0
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (3) d = 1 - S0 * y0 in f7
  (p6) fnma.s1 f7=f6,f7,f1
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (4) T0 = d * S0 in f8
  (p6) fma.s1 f8=f7,f6,f0
  nop.i 0
} { .mfi
  nop.m 0
  // Step (5) e = 1/2 + 3/8 * d in f7
  (p6) fma.s1 f7=f7,f10,f9
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (6) S = S0 + e * T0 in f7
  (p6) fma.s.s0 f7=f7,f8,f6
  nop.i 0;;
}
```

```
  // END NON-IEEE SINGLE PRECISION SQUARE ROOT ALGORITHM

{ .mib
  // store result
  stfs [r33]=f7
  nop.i 0
  // return
  br.ret.sptk b0;;
}
```

.endp sgl_sqrt

Sample test driver:

```
#include <stdio.h>
void sgl_sqrt (float *, float *);

void run_test (unsigned int ia) {
  float a, q, q0;
  unsigned int iq;

  *(unsigned int *)(&a) = ia;
  q0 = sqrtf (a);
  iq = *(unsigned int *)(&q0);

  sgl_sqrt (&a, &q);

  printf ("\nArgument: %lx\nSquare Root: %lx\n", ia, iq);
  if (iq == *(unsigned int *)(&q)) {
    printf ("Passed\n");
  } else {
    printf ("Failed: q = %8.8x\n", *(unsigned int *)(&q));
    exit (1);
  }
}

void main () {
  run_test (0x00000000); // sqrt (+0.0)
  run_test (0x80000000); // sqrt (-0.0)
  run_test (0xbf800000); // sqrt (-1.0)
  run_test (0x7f800000); // sqrt (+inf)
  run_test (0xff800000); // sqrt (-inf)
  run_test (0x3f800000); // sqrt (+1.0)
  run_test (0x3f800001); // sqrt (1+1u)
  run_test (0x3f800002); // sqrt (1+2u)
  run_test (0x3f87676a);
  run_test (0x3f8bca73);
  printf ("\n\tALL TESTS PASSED\n");
}
```

3.2. Double Precision Floating-Point Square Root, Non-IEEE

This algorithm calculates $S = \sqrt{a}$ in double precision, where *a* is a double precision number. *rn* is the IEEE round-to-nearest mode, and *rnd* is any IEEE rounding mode. All other symbols used are 82-bit floating-point register format numbers. The precision used for each step is shown below. The result has an error of less than 0.5001 ulp, but most often of less than 0.5 ulp. (Algorithm derived from Doc. Nr. 248725-004, section 3.4)

Itanium 2 latency: 32 clock cycles (36 clock cycles for IEEE-correct square root)

Itanium 2 throughput: 5.5 clock cycles/operation (6.5 clock cycles/operation for IEEE-correct square root)

(1)	$y_0 = \left(1 / \sqrt{a}\right) \cdot (1+\varepsilon_0),\ \ |\varepsilon_0| < 2^{-8.831}$	table lookup
(2)	$H_0 = (0.5 \cdot y_0)_{rn}$	82-bit floating-point register format
(3)	$G_0 = (a \cdot y_0)_{rn}$	82-bit floating-point register format
(4)	$r_0 = (0.5 - G_0 \cdot H_0)_{rn}$	82-bit floating-point register format
(5)	$H_1 = (H_0 + r_0 \cdot H_0)_{rn}$	82-bit floating-point register format
(6)	$G_1 = (G_0 + r_0 \cdot G_0)_{rn}$	82-bit floating-point register format
(7)	$r_1 = (0.5 - G_1 \cdot H_1)_{rn}$	82-bit floating-point register format
(8)	$H_2 = (H_1 + r_1 \cdot H_1)_{rn}$	82-bit floating-point register format
(9)	$G_2 = (G_1 + r_1 \cdot G_1)_{rn}$	82-bit floating-point register format
(10)	$d_2 = (a - G_2 \cdot G_2)_{rn}$	82-bit floating-point register format
(11)	$S = (G_2 + d_2 \cdot H_2)_{rnd}$	double precision

The assembly language implementation:

```
  .file "dbl_sqrt.s"
  .section .text
  .proc dbl_sqrt#
  .global dbl_sqrt#
  .align 32

dbl_sqrt:

{ .mmi
  alloc r31=ar.pfs,2,0,0,0  // r32, r33

  // &a is in r32
  // &sqrt is in r33 (the address of the sqrt result)

  // load the argument a in f6
  ldfd f6 = [r32]
  nop.i 0
}

  // BEGIN NON-IEEE DOUBLE PRECISION SQUARE ROOT ALGORITHM
  // general registers used: r2, r31, r32, r33
  // predicate registers used: p6
  // floating-point registers used: f6, f7, f8, f9, f10

{ .mmi
  // exponent of +1/2 in r2
  mov r2 = 0x0fffe;;
  // +1/2 in f10
  setf.exp f10 = r2
  nop.i 0
} { .mfi
  nop.m 0
  // Step (1) y0 = 1/sqrt(a) in f7
  frsqrta.s0 f7,p6=f6
  nop.i 0;;
} { .mfi
```

```
  nop.m 0
  // Step (2) H0 = +1/2 * y0 in f8
  (p6) fma.s1 f8=f10,f7,f0
  nop.i 0
} { .mfi
  nop.m 0
  // Step (3) G0 = a * y0 in f7
  (p6) fma.s1 f7=f6,f7,f0
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (4) r0 = 1/2 - G0 * H0 in f9
  (p6) fnma.s1 f9=f7,f8,f10
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (5) H1 = H0 + r0 * H0 in f8
  (p6) fma.s1 f8=f9,f8,f8
  nop.i 0
} { .mfi
  nop.m 0
  // Step (6) G1 = G0 + r0 * G0 in f7
  (p6) fma.s1 f7=f9,f7,f7
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (7) r1 = 1/2 - G1 * H1 in f9
  (p6) fnma.s1 f9=f7,f8,f10
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (8) H2 = H1 + r1 * H1 in f8
  (p6) fma.s1 f8=f9,f8,f8
  nop.i 0
} { .mfi
  nop.m 0
  // Step (9) G2 = G1 + r1 * G1 in f7
  (p6) fma.s1 f7=f9,f7,f7
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (10) d2 = a - G2 * G2 in f9
  (p6) fnma.s1 f9=f7,f7,f6
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (11) S = G2 + d2 * H2 in f7
  (p6) fma.d.s0 f7=f9,f8,f7
  nop.i 0;;
}

  // END NON-IEEE DOUBLE PRECISION SQUARE ROOT ALGORITHM

{ .mib
  // store result
  stfd [r33]=f7
  nop.i 0
  // return
  br.ret.sptk b0;;
}

  .endp dbl_sqrt
```

Sample test driver:

```
#include <stdio.h>
void dbl_sqrt (double *, double *);
```

```
void run_test (unsigned __int64 ia) {
  double a, q, q0;
  unsigned __int64 iq;

  *(unsigned __int64 *)(&a) = ia;
  q0 = sqrt (a);
  iq = *(unsigned __int64 *)(&q0);

  dbl_sqrt (&a, &q);

  printf ("\nArgument: %I64x\nSquare Root: %I64x\n", ia, iq);
  if (iq == *(unsigned __int64 *)(&q)) {
    printf ("Passed\n");
  } else {
    printf ("Failed: q = %I64x\n", *(unsigned __int64 *)(&q));
    exit (1);
  }
}

void main () {
  run_test (0x0000000000000000); // sqrt(+0.0)
  run_test (0x8000000000000000); // sqrt(-0.0)
  run_test (0x3ff0000000000000); // sqrt(+1.0)
  run_test (0xbff0000000000000); // sqrt(-1.0)
  run_test (0x7ff0000000000000); // sqrt(+inf)
  run_test (0xfff0000000000000); // sqrt(-inf)
  run_test (0x3ff0000000000001); // sqrt(1+1u)
  run_test (0x3ff0000000000002); // sqrt(1+2u)
  run_test (0x3f8e676a8878987b);
  run_test (0x3acff9de398fc87d);
  printf ("\n\tALL TESTS PASSED\n");
}
```

4. Non-IEEE Floating-Point Reciprocal

Two main algorithms are provided: one for single precision, and one for double precision reciprocal operations. For single precision reciprocal operations, an alternate algorithm is provided as well that has better accuracy, but requires one extra floating-point instruction to implement. Its latency is the same as that of the main algorithm, but the extra instruction makes it perform slightly worse if used as a throughput-optimized algorithm in software-pipelined loops.

4.1. Single Precision Floating-Point Reciprocal, Non-IEEE

The following algorithm calculates $q = 1/b$ in single precision, where b is a single precision number. *rn* is the IEEE round-to-nearest mode, and *rnd* is any IEEE rounding mode. All other symbols used are 82-bit floating-point register format numbers. The precision used for each step is shown below. The result has an error of less than 0.6585 ulp, but most often of less than 0.5 ulp. (Algorithm derived from Doc. Nr. 248725-004, section 2.2)

Itanium 2 latency: 16 clock cycles – same as the alternate algorithm (latency is of 24 clock cycles for IEEE-correct reciprocal, using the latency-optimized single precision division algorithm)

Itanium 2 throughput: 2.0 clock cycles/operation (3.5 clock cycles/operation for an IEEE-correct reciprocal, using the throughput-optimized single precision division algorithm)

(1)	$y_0 = 1 / b \cdot (1+\varepsilon_0),\ \ \lvert\varepsilon_0\rvert<2^{-8.886}$	table lookup
(2)	$d = (1 - b \cdot y_0)_{rn}$	82-bit floating-point register format
(3)	$e = (d \cdot d + d)_{rn}$	82-bit floating-point register format
(4)	$y_1 = (y_0 + e \cdot y_0)_{rnd}$	single precision

The assembly language implementation:

```
  .file "sgl_recip.s"
  .section .text
  .proc sgl_recip#
  .global sgl_recip#
  .align 32

sgl_recip:

{ .mmi
  alloc r31=ar.pfs,2,0,0,0  // r32, r33

  // &b is in r32
  // &recip is in r33 (the address of the reciprocal result)

  // load the argument b in f6
  ldfs f6 = [r32]
  nop.i 0;;
}

  // BEGIN NON-IEEE SINGLE PRECISION RECIPROCAL ALGORITHM
  // general registers used: none
  // predicate registers used: p6
  // floating-point registers used: f6, f7

{ .mfi
  nop.m 0
  // Step (1) y0 = 1 / b in f6
  frcpa.s0 f7,p6=f1,f6
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (2) d = 1 - b * y0 in f6
  (p6) fnma.s1 f6=f6,f7,f1
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (3) e = d * d + d in f6
  (p6) fma.s1 f6=f6,f6,f6
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (4) y1 = y0 + e * y0 in f7
  (p6) fma.s.s0 f7=f7,f6,f7
  nop.i 0;;
}

  // END NON-IEEE SINGLE PRECISION RECIPROCAL ALGORITHM
```

```
{ .mib
  // store result
  stfs [r33]=f7
  nop.i 0
  // return
  br.ret.sptk b0;;
}

  .endp sgl_recip
```

Sample test driver:

```
#include <stdio.h>
void sgl_recip (float *, float *);

void run_test (unsigned int ib) {
  float b, q, q0;
  unsigned int iq;

  *(unsigned int *)(&b) = ib;
  q0 = 1 / b;
  iq = *(unsigned int *)(&q0);

  sgl_recip (&b, &q);

  printf ("Argument: %lx\nReciprocal: %lx\n", ib, iq);
  if (iq == *(unsigned int *)(&q)) {
    printf ("Passed\n");
  } else {
    printf ("Failed: q = %8.8x\n", *(unsigned int *)(&q));
    exit (1);
  }
}

void main () {
  run_test (0x00000000); // 1 / (+0.0)
  run_test (0x80000000); // 1 / (-0.0)
  run_test (0xbf800000); // 1 / (-1.0)
  run_test (0xff800000); // 1 / (-inf)
  run_test (0x3f800000); // 1 / (+1.0)
  run_test (0x3f800001); // 1 / (1+1u)
  run_test (0x3f800003);
  run_test (0x3f87676a);
  run_test (0x3f8ffffe);
  run_test (0x3f8fffff);
  printf ("\n\tALL TESTS PASSED\n");
}
```

Alternate Algorithm

The following algorithm calculates $q = 1/b$ in single precision, where b is a single precision number. *rn* is the IEEE round-to-nearest mode, and *rnd* is any IEEE rounding mode. All other symbols used are 82-bit floating-point register format numbers. The precision used for each step is shown below. The result has an error of less than 0.5004 ulp, but most often of less than 0.5 ulp. (Algorithm derived from Doc. Nr. 248725-004, section 2.4)

Itanium 2 latency: 16 clock cycles – same as that of the previous algorithm (24 clock cycles for IEEE-correct reciprocal, using the latency-optimized single precision division algorithm)

Itanium 2 throughput: 2.5 clock cycles/operation (throughput is of 3.5 clock cycles/operation for IEEE-correct reciprocal, using the throughput-optimized single precision division algorithm)

(1)	$y_0 = 1 / b \cdot (1+\varepsilon_0),\ \ |\varepsilon_0| < 2^{-8.886}$	table lookup
(2)	$e_0 = (1 - b \cdot y_0)_{rn}$	82-bit floating-point register format
(3)	$y_1 = (y_0 + e_0 \cdot y_0)_{rn}$	82-bit floating-point register format
(4)	$e_1 = (e_0 \cdot e_0)_{rn}$	82-bit floating-point register format
(5)	$y_2 = (y_1 + e_1 \cdot y_1)_{rnd}$	single precision

The assembly language implementation:

```
  .file "sgl_recip1.s"
  .section .text
  .proc sgl_recip1#
  .global sgl_recip1#
  .align 32

sgl_recip1:

{ .mmi
  alloc r31=ar.pfs,2,0,0,0  // r32, r33

  // &b is in r32
  // &recip is in r33 (the address of the reciprocal result)

  // load the argument b in f6
  ldfs f6 = [r32]
  nop.i 0;;
}

  // BEGIN NON-IEEE SINGLE PRECISION RECIPROCAL ALGORITHM
  // general registers used: none
  // predicate registers used: p6
  // floating-point registers used: f6, f7


{ .mfi
  nop.m 0
  // Step (1) y0 = 1 / b in f7
  frcpa.s0 f7,p6=f1,f6
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (2) e0 = 1 - b * y0 in f6
  (p6) fnma.s1 f6=f6,f7,f1
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (3) y1 = y0 + e0 * y0 in f7
  (p6) fma.s1 f7=f7,f6,f7
  nop.i 0
} { .mfi
  nop.m 0
  // Step (4) e1 = e0 * e0 in f6
  (p6) fma.s1 f6=f6,f6,f0
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (5) y2 = y1 + e1 * y1 in f7
  (p6) fma.s.s0 f7=f7,f6,f7
  nop.i 0;;
}

  // END NON-IEEE SINGLE PRECISION RECIPROCAL ALGORITHM
```

```
{ .mib
  // store result
  stfs [r33]=f7
  nop.i 0
  // return
  br.ret.sptk b0;;
}

  .endp sgl_recip1
```

4.2. Double Precision Floating-Point Reciprocal, Non-IEEE

This algorithm calculates $q = 1/b$ in double precision, where b is a double precision number. *rn* is the IEEE round-to-nearest mode, and *rnd* is any IEEE rounding mode. All other symbols used are 82-bit floating-point register format numbers. The precision used for each step is shown below. The result has an error of less than 0.5010 ulp, but often of less than 0.5 ulp. (Algorithm derived from Doc. Nr. 248725-004, section 2.4)

Itanium 2 latency: 20 clock cycles (28 clock cycles for IEEE-correct reciprocal, using the latency-optimized double precision division algorithm)

Itanium 2 throughput: 3.5 clock cycles/operation (5.0 clock cycles/operation for IEEE-correct reciprocal, using the throughput-optimized double precision division algorithm)

(1)	$y_0 = 1 / b \cdot (1+\varepsilon_0),\ \ \lvert\varepsilon_0\rvert<2^{-8.886}$	table lookup
(2)	$e_0 = (1 - b \cdot y_0)_{rn}$	82-bit floating-point register format
(3)	$y_1 = (y_0 + e_0 \cdot y_0)_{rn}$	82-bit floating-point register format
(4)	$e_1 = (e_0 \cdot e_0)_{rn}$	82-bit floating-point register format
(5)	$y_2 = (y_1 + e_1 \cdot y_1)_{rn}$	82-bit floating-point register format
(6)	$e_2 = (e_1 \cdot e_1)_{rn}$	82-bit floating-point register format
(7)	$y_3 = (y_2 + e_2 \cdot y_2)_{rnd}$	double precision

The assembly language implementation:

```
  .file "dbl_recip.s"
  .section .text
  .proc dbl_recip#
  .global dbl_recip#
  .align 32

dbl_recip:

{ .mmi
  alloc r31=ar.pfs,2,0,0,0  // r32, r33

  // &b is in r32
  // &recip is in r33 (the address of the reciprocal result)

  // load the argument b in f6
  ldfd f6 = [r32]
```

```
  nop.i 0;;
}

  // BEGIN NON-IEEE DOUBLE PRECISION RECIPROCAL ALGORITHM
  // general registers used: none
  // predicate registers used: p6
  // floating-point registers used: f6, f7

 { .mfi
  nop.m 0
  // Step (1) y0 = 1 / b in f7
  frcpa.s0 f7,p6=f1,f6
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (2) e0 = 1 - b * y0 in f6
  (p6) fnma.s1 f6=f6,f7,f1
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (3) y1 = y0 + e0 * y0 in f7
  (p6) fma.s1 f7=f7,f6,f7
  nop.i 0
} { .mfi
  nop.m 0
  // Step (4) e1 = e0 * e0 in f6
  (p6) fma.s1 f6=f6,f6,f0
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (5) y2 = y1 + e1 * y1 in f7
  (p6) fma.s1 f7=f7,f6,f7
  nop.i 0
} { .mfi
  nop.m 0
  // Step (6) e2 = e1 * e1 in f6
  (p6) fma.s1 f6=f6,f6,f0
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (7) y3 = y2 + e2 * y2 in f7
  (p6) fma.d.s0 f7=f7,f6,f7
  nop.i 0;;
}

  // END NON-IEEE DOUBLE PRECISION RECIPROCAL ALGORITHM

{ .mib
  // store result
  stfd [r33]=f7
  nop.i 0
  // return
  br.ret.sptk b0;;
}

  .endp dbl_recip
```

Sample test driver:

```
#include <stdio.h>
void dbl_recip (double *, double *);

void run_test (unsigned __int64 ib) {
  double b, q, q0;
  unsigned __int64 iq;

  *(unsigned __int64 *)(&b) = ib;
```

```
  q0 = 1 / b;
  iq = *(unsigned __int64 *)(&q0);

  dbl_recip (&b, &q);

  printf ("Argument: %I64x\nReciprocal: %I64x\n", ib, iq);
  if (iq == *(unsigned __int64 *)(&q)) {
    printf ("Passed\n");
  } else {
    printf ("Failed: q = %I64x\n", *(unsigned __int64 *)(&q));
    exit (1);
  }
}

void main () {
  run_test (0x0000000000000000); // 1 / (+0.0)
  run_test (0x8000000000000000); // 1 / (-0.0)
  run_test (0xbff0000000000000); // 1 / (-1.0)
  run_test (0xfff0000000000000); // 1 / (-inf)
  run_test (0x3ff0000000000001); // 1 / (1+1u)
  run_test (0x3ff0000000000002); // 1 / (1+2u)
  run_test (0x3ff0000000000003);
  run_test (0x3fffffffffffffe);
  run_test (0x3fffffffffffffd);
  run_test (0x3fffffffffffffff);
  printf ("\n\tALL TESTS PASSED\n");
}
```

5. Non-IEEE Floating-Point Reciprocal Square Root

Two algorithms are provided: one for single precision, and one for double precision reciprocal square root operations. In both cases, an alternate implementation is shown as well, that handles in a different way the special cases of negative, zero, infinity, and NaN operands.

5.1. Single Precision Floating-Point Reciprocal Square Root, Non-IEEE

This algorithm calculates $S = \left(1 / \sqrt{a}\right)$ in single precision, where a is a single precision number. *rn* is the IEEE round-to-nearest mode, and *rnd* is any IEEE rounding mode. All other symbols used are 82-bit floating-point register format numbers. The precision used for each step is shown below. The result has an error of less than 0.9449 ulp, but most often of less than 0.5 ulp. (Algorithm derived from Doc. Nr. 248725-004, section 3.1)

Itanium 2 latency: 20 clock cycles (52 clock cycles for IEEE-correct square root followed by IEEE-correct division, using the latency-optimized single precision algorithms)

Itanium 2 throughput: 3.5 clock cycles/operation, with 0.5 clock cycles due to one extra **frcpa** (8.5 clock cycles/operation for IEEE-correct square root followed by IEEE-correct division, using the throughput-optimized double precision algorithms)

(1)	$y_0 = \left(1 / \sqrt{a}\right) \cdot (1+\varepsilon_0),\ \ \lvert\varepsilon_0\rvert < 2^{-8.831}$	table lookup
(2)	$S_0 = (a \cdot y_0)_{rn}$	82-bit floating-point register format
(3)	$d = (1 - S_0 \cdot y_0)_{rn}$	82-bit floating-point register format
(4)	$e = (0.5 + 0.375 \cdot d)_{rn}$	82-bit floating-point register format
(5)	$T_0 = (d \cdot y_0)_{rn}$	82-bit floating-point register format
(6)	$S = (y_0 + e \cdot T_0)_{rnd}$	single precision

The assembly language implementation:

```
  .file "sgl_rsqrt.s"
  .section .text
  .proc sgl_rsqrt#
  .global sgl_rsqrt#
  .align 32

sgl_rsqrt:

{ .mmi
  alloc r31=ar.pfs,2,0,0,0  // r32, r33

  // &a is in r32
  // &sqrt is in r33 (the address of the sqrt result)

  // load the argument a in f6
  ldfs f6 = [r32]
  nop.i 0;;
}

  // BEGIN NON-IEEE SINGLE PRECISION RECIPROCAL SQUARE ROOT ALGORITHM
  // general registers used: r2
  // predicate registers used: p6
  // floating-point registers used: f6, f8, f7, f9

{ .mlx
  nop.m 0
  // +1/2 in f8
  movl r2 = 0x3f000000;;
} { .mlx
  setf.s f8 = r2
  // +3/8 in f9
  movl r2=0x3ec00000;;
} { .mfi
  setf.s f9 = r2

  // Step (1) y0 = 1/sqrt(a) in f7
  frsqrta.s0 f7,p6=f6
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (2) S0 = a * y0 in f6
  (p6) fma.s1 f6=f6,f7,f0
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (3) d = 1 - S0 * y0 in f6
  (p6) fnma.s1 f6=f6,f7,f1
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (4) e = 1/2 + 3/8 * d in f8
  (p6) fma.s1 f8=f9,f6,f8
```

```
  nop.i 0
} { .mfi
  nop.m 0
  // Step (5) T0 = d * y0 in f6
  (p6) fma.s1 f6=f6,f7,f0
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (6) y = y0 + e * T0 in f6
  (p6) fma.s.s0 f7=f6,f8,f7
  nop.i 0;;
}

  // END NON-IEEE SINGLE PRECISION RECIPROCAL SQUARE ROOT ALGORITHM

{ .mib
  // store result
  (p6) stfs [r33]=f7
  nop.i 0
  // return
  (p6) br.ret.sptk b0
}

{ .mfi
  nop.m 0
  // inverse of frsqrta result for negative, zero, infinity, or NaN operand
  frcpa.s0 f7,p0=f1,f7
  nop.i 0;;
}

  // END NON-IEEE DOUBLE PRECISION RECIPROCAL SQUARE ROOT (SPECIAL ARGUMENT)

{ .mib
  stfs [r33]=f7 // store result
  nop.i 0
  br.ret.sptk b0;; // return
}

  .endp sgl_rsqrt
```

The last **frcpa**, applied to the result of the **frsqrta** instruction, is needed to generate the correct result when the operand is negative, zero, infinity, or NaN. In these cases predicate register `p6` is cleared, and the iterative computation is predicated off. The result should be the inverse of the result from **frsqrta**, and the inversion is performed by **frcpa**.

Alternate Implementation

An alternate implementation that handles differently the special cases of negative, zero, infinity, or NaN operands is shown next. It is based on calculating in `p7` the negation of the predicate `p6` which is written by the **frsqrta** instruction. General register `r3` is also used for this purpose:

```
  .file "sgl_rsqrt1.s"
  .section .text
  .proc sgl_rsqrt1#
  .global sgl_rsqrt1#
  .align 32

sgl_rsqrt1:

{ .mmi
  alloc r31=ar.pfs,2,0,0,0  // r32, r33
```

```
  // &a is in r32
  // &sqrt is in r33 (the address of the sqrt result)

  // load the argument a in f6
  ldfs f6 = [r32]
  nop.i 0;;
}

  // BEGIN NON-IEEE SINGLE PRECISION RECIPROCAL SQUARE ROOT ALGORITHM
  // general registers used: r2, r3
  // predicate registers used: p6, p7
  // floating-point registers used: f6, f8, f7, f9

{ .mlx
  add r3=0,r0
  // +1/2 in f8
  movl r2 = 0x3f000000;;
} { .mlx
  setf.s f8 = r2
  // +3/8 in f9
  movl r2=0x3ec00000;;
} { .mfi
  setf.s f9 = r2

  // Step (1) y0 = 1/sqrt(a) in f7
  frsqrta.s0 f7,p6=f6
  nop.i 0;;
} { .mfi
  (p6) add r3=1,r0
  // Step (2) S0 = a * y0 in f6
  (p6) fma.s1 f6=f6,f7,f0
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (3) d = 1 - S0 * y0 in f6
  (p6) fnma.s1 f6=f6,f7,f1
  cmp.eq.unc p7,p0=r3,r0;;
} { .mfb
  nop.m 0
  // Step (4) e = 1/2 + 3/8 * d in f8
  (p6) fma.s1 f8=f9,f6,f8
  (p7) br.cond.dpnt frcpa_lbl
} { .mfi
  nop.m 0
  // Step (5) T0 = d * y0 in f6
  (p6) fma.s1 f6=f6,f7,f0
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (6) y = y0 + e * T0 in f6
  (p6) fma.s.s0 f7=f6,f8,f7
  nop.i 0;;
}

  // END NON-IEEE SINGLE PRECISION RECIPROCAL SQUARE ROOT ALGORITHM

stfs_lbl:
{ .mib
  stfs [r33]=f7 // store result
  nop.i 0
  br.ret.sptk b0;; // return
}

frcpa_lbl:
{ .mfb
  nop.m 0
  // inverse of frsqrta result for negative, zero, infinity, or NaN operand
```

```
  frcpa.s0 f7,p0=f1,f7
  br.cond.sptk stfs_lbl;;
}

  .endp sgl_rsqrt1
```

The new instructions can be scheduled so as not to affect the latency of the algorithm on its main path.

The complement `p7` of the predicate `p6` written by **frsqrta** could be obtained also with:

```
  cmp.eq p7,p0=r0,r0;; // p7=1
  (p6) cmp.ne p7,p0=r0,r0 // p7=not(p6)
```

Sample test driver:

```
#include <stdio.h>
void sgl_rsqrt (float *, float *);

void run_test (unsigned int ia) {
  float a, q, q0;
  unsigned int iq;

  *(unsigned int *)(&a) = ia;
  q0 = 1.0 / sqrtf (a);
  iq = *(unsigned int *)(&q0);

  sgl_rsqrt (&a, &q);

  printf ("\nArgument: %lx\nSquare Root: %lx\n", ia, iq);
  if (iq == *(unsigned int *)(&q)) {
    printf ("Passed\n");
  } else {
    printf ("Failed: q = %8.8x\n", *(unsigned int *)(&q));
    exit (1);
  }
}

void main () {
  run_test (0x7f800001); // rsqrt (SNaN)
  run_test (0x7fc00001); // rsqrt (QNaN)
  run_test (0xff800000); // rsqrt (-inf)
  run_test (0xbf800000); // rsqrt (-1.0)
  run_test (0x80000000); // rsqrt (-0.0)
  run_test (0x00000000); // rsqrt (+0.0)
  run_test (0x00000fff); // rsqrt (+den)
  run_test (0x3f800000); // rsqrt (+1.0)
  run_test (0x3f800002); // rsqrt (1+2u)
  run_test (0x3f8fc87d);
  printf ("\n\tALL TESTS PASSED\n");
}
```

5.2. Double Precision Floating-Point Reciprocal Square Root, Non-IEEE

This algorithm calculates $S = \left(1 / \sqrt{a}\right)$ in double precision, where a is a double precision number. *rn* is the IEEE round-to-nearest mode, and *rnd* is any IEEE rounding mode. All other symbols used are 82-bit floating-point register format numbers. The precision used

for each step is shown below. The result has an error of less than 0.5031 ulp, but most often of less than 0.5 ulp. (Algorithm derived from Doc. Nr. 248725-004, section 3.4)

Itanium 2 latency: 32 clock cycles (64 clock cycles for IEEE-correct square root followed by IEEE-correct division, using the latency-optimized double precision algorithms)

Itanium 2 throughput: 6.5 clock cycles/operation, with 0.5 clock cycles due to one extra **frcpa** (11.5 clock cycles/operation for IEEE-correct square root followed by IEEE-correct division, using the throughput-optimized double precision algorithms)

(1)	$y_0 = \left(1 / \sqrt{a}\right) \cdot (1+\varepsilon_0),\ \ \lvert\varepsilon_0\rvert<2^{-8.831}$	table lookup
(2)	$H_0 = (0.5 \cdot y_0)_{rn}$	82-bit floating-point register format
(3)	$G_0 = (a \cdot y_0)_{rn}$	82-bit floating-point register format
(4)	$r_0 = (0.5 - G_0 \cdot H_0)_{rn}$	82-bit floating-point register format
(5)	$H_1 = (H_0 + r_0 \cdot H_0)_{rn}$	82-bit floating-point register format
(6)	$G_1 = (G_0 + r_0 \cdot G_0)_{rn}$	82-bit floating-point register format
(7)	$r_1 = (0.5 - G_1 \cdot H_1)_{rn}$	82-bit floating-point register format
(8)	$H_2 = (H_1 + r_1 \cdot H_1)_{rn}$	82-bit floating-point register format
(9)	$G_2 = (G_1 + r_1 \cdot G_1)_{rn}$	82-bit floating-point register format
(10)	$r_2 = (0.5 - G_2 \cdot H_2)_{rn}$	82-bit floating-point register format
(11)	$H_2' = (H_2 + H_2)_{rn}$	82-bit floating-point register format
(12)	$S = (H_2' + r_2 \cdot H_2')_{rnd}$	double precision

The assembly language implementation:

```
  .file "dbl_rsqrt.s"
  .section .text
  .proc dbl_rsqrt#
  .global dbl_rsqrt#
  .align 32

dbl_rsqrt:

{ .mmi
  alloc r31=ar.pfs,2,0,0,0  // r32, r33

  // &a is in r32
  // &sqrt is in r33 (the address of the sqrt result)

  // load the argument a in f6
  ldfd f6 = [r32]
  nop.i 0;;
}

  // BEGIN NON-IEEE DOUBLE PRECISION RECIPROCAL SQUARE ROOT ALGORITHM
  // general registers used: r2
  // predicate registers used: p6
  // floating-point registers used: f6, f7, f8, f9, f10


{ .mmi
  // exponent of +1/2 in r2
  mov r2 = 0x0fffe;;
```

```
  // +1/2 in f10
  setf.exp f10 = r2
  nop.i 0
} { .mfi
  nop.m 0
  // Step (1) y0 = 1/sqrt(a) in f7
  frsqrta.s0 f7,p6=f6
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (2) H0 = +1/2 * y0 in f8
  (p6) fma.s1 f8=f10,f7,f0
  nop.i 0
} { .mfi
  nop.m 0
  // Step (3) G0 = a * y0 in f7
  (p6) fma.s1 f7=f6,f7,f0
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (4) r0 = 1/2 - G0 * H0 in f9
  (p6) fnma.s1 f9=f7,f8,f10
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (5) H1 = H0 + r0 * H0 in f8
  (p6) fma.s1 f8=f9,f8,f8
  nop.i 0
} { .mfi
  nop.m 0
  // Step (6) G1 = G0 + r0 * G0 in f7
  (p6) fma.s1 f7=f9,f7,f7
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (7) r1 = 1/2 - G1 * H1 in f9
  (p6) fnma.s1 f9=f7,f8,f10
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (8) H2 = H1 + r1 * H1 in f8
  (p6) fma.s1 f8=f9,f8,f8
  nop.i 0
} { .mfi
  nop.m 0
  // Step (9) G2 = G1 + r1 * G1 in f7
  (p6) fma.s1 f7=f9,f7,f7
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (10) r2 = 1/2 - G2 * H2 in f9
  (p6) fnma.s1 f9=f7,f8,f10
  nop.i 0
} { .mfi
  nop.m 0
  // Step (11) H2' = H2 + H2 in f8
  (p6) fadd.s1 f8=f8,f8
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (12) S = H2' + r2 * H2' in f7
  (p6) fma.d.s0 f7=f8,f9,f8
  nop.i 0;;
}

  // END NON-IEEE DOUBLE PRECISION RECIPROCAL SQUARE ROOT ALGORITHM
```

```
{ .mib
  (p6) stfd [r33]=f7 // store result
  nop.i 0
  (p6) br.ret.sptk b0 // return
}

{ .mfi
  nop.m 0
  // inverse of frsqrta result for negative, zero, infinity, or NaN operand
  frcpa.s0 f7,p0=f1,f7
  nop.i 0;;
}

  // END NON-IEEE DOUBLE PRECISION RECIPROCAL SQUARE ROOT (SPECIAL ARGUMENT)

{ .mib
  stfd [r33]=f7 // store result
  nop.i 0
  br.ret.sptk b0;; // return
}
```

.endp dbl_rsqrt

The last **frcpa**, applied to the result of the **frsqrta** instruction, is needed to generate the correct result when the operand is negative, zero, infinity, or NaN. In these cases predicate register `p6` is cleared, and the iterative computation is predicated off. The result should be the inverse of the result from **frsqrta**, and the inversion is performed by **frcpa**.

Alternate Implementation

An alternate implementation that handles differently the special cases of negative, zero, infinity, or NaN operands is shown next. It is based on calculating in `p7` the negation of the predicate `p6` which is written by the **frsqrta** instruction. General register `r3` is also used for this purpose:

```
  .file "dbl_rsqrt1.s"
  .section .text
  .proc dbl_rsqrt1#
  .global dbl_rsqrt1#
  .align 32

dbl_rsqrt1:

{ .mmi
  alloc r31=ar.pfs,2,0,0,0  // r32, r33

  // &a is in r32
  // &sqrt is in r33 (the address of the sqrt result)

  // load the argument a in f6
  ldfd f6 = [r32]
  nop.i 0;;
}

  // BEGIN NON-IEEE DOUBLE PRECISION RECIPROCAL SQUARE ROOT ALGORITHM
  // general registers used: r2, r3
  // predicate registers used: p6, p7
  // floating-point registers used: f6, f7, f8, f9, f10

{ .mmi
  // exponent of +1/2 in r2
  mov r2 = 0x0fffe;;
```

```
  // +1/2 in f10
  setf.exp f10 = r2
  add r3=0,r0
} { .mfi
  nop.m 0
  // Step (1) y0 = 1/sqrt(a) in f7
  frsqrta.s0 f7,p6=f6
  nop.i 0;;
} { .mfi
  (p6) add r3=1,r0
  // Step (2) H0 = +1/2 * y0 in f8
  (p6) fma.s1 f8=f10,f7,f0
  nop.i 0
} { .mfi
  nop.m 0
  // Step (3) G0 = a * y0 in f7
  (p6) fma.s1 f7=f6,f7,f0
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (4) r0 = 1/2 - G0 * H0 in f9
  (p6) fnma.s1 f9=f7,f8,f10
  cmp.eq.unc p7,p0=r3,r0;;
} { .mfi
  nop.m 0
  // Step (5) H1 = H0 + r0 * H0 in f8
  (p6) fma.s1 f8=f9,f8,f8
  nop.i 0
} { .mfb
  nop.m 0
  // Step (6) G1 = G0 + r0 * G0 in f7
  (p6) fma.s1 f7=f9,f7,f7
  (p7) br.cond.dpnt frcpa_lbl;;
} { .mfi
  nop.m 0
  // Step (7) r1 = 1/2 - G1 * H1 in f9
  (p6) fnma.s1 f9=f7,f8,f10
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (8) H2 = H1 + r1 * H1 in f8
  (p6) fma.s1 f8=f9,f8,f8
  nop.i 0
} { .mfi
  nop.m 0
  // Step (9) G2 = G1 + r1 * G1 in f7
  (p6) fma.s1 f7=f9,f7,f7
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (10) r2 = 1/2 - G2 * H2 in f9
  (p6) fnma.s1 f9=f7,f8,f10
  nop.i 0
} { .mfi
  nop.m 0
  // Step (11) H2' = H2 + H2 in f8
  (p6) fadd.s1 f8=f8,f8
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (12) S = H2' + r2 * H2' in f7
  (p6) fma.d.s0 f7=f8,f9,f8
  nop.i 0;;
}

  // END NON-IEEE DOUBLE PRECISION RECIPROCAL SQUARE ROOT ALGORITHM
```

```
stfd_lbl:
{ .mib
  stfd [r33]=f7 // store result
  nop.i 0
  br.ret.sptk b0;; // return
}

frcpa_lbl:
{ .mfb
  nop.m 0
  // inverse of frsqrta result for negative, zero, infinity, or NaN operand
  frcpa.s0 f7,p0=f1,f7
  br.cond.sptk stfd_lbl;;
}

  .endp dbl_rsqrt1
```

The new instructions can be scheduled so as not to affect the latency of the algorithm on its main path.

The complement `p7` of the predicate `p6` written by **frsqrta** could be obtained also with:

```
  cmp.eq p7,p0=r0,r0;; // p7=1
  (p6) cmp.ne p7,p0=r0,r0 // p7=not(p6)
```

Sample test driver:

```
#include <stdio.h>
void dbl_rsqrt (double *, double *);

void run_test (unsigned __int64 ia) {
  double a, q, q0;
  unsigned __int64 iq;

  *(unsigned __int64 *)(&a) = ia;
  q0 = 1.0 / sqrt (a);
  iq = *(unsigned __int64 *)(&q0);

  dbl_rsqrt (&a, &q);

  printf ("\nArgument: %I64x\nSquare Root: %I64x\n", ia, iq);
  if (iq == *(unsigned __int64 *)(&q)) {
    printf ("Passed\n");
  } else {
    printf ("Failed: q = %I64x\n", *(unsigned __int64 *)(&q));
    exit (1);
  }
}


void main () {
  run_test (0x7ff0000000000001); // rsqrt (SNaN)
  run_test (0x7ff8000000000001); // rsqrt (QNaN)
  run_test (0xbff0000000000000); // rsqrt (-1.0)
  run_test (0x8000000000000000); // rsqrt (-0.0)
  run_test (0x0000000000000000); // rsqrt (+0.0)
  run_test (0x7ff0000000000000); // rsqrt (+inf)
  run_test (0x0000000000000fff); // rsqrt (+den)
  run_test (0x3ff0000000000000); // rsqrt (+1.0)
  run_test (0x3ff0000000000002); // rsqrt (1+2u)
  run_test (0x3f8e676a8878987b);
  printf ("\n\tALL TESTS PASSED\n");
}
```

6. Authors

Marius Cornea, John Harrison, Cristina Iordache, Bob Norin, Shane Story

7. Acknowledgments

The authors wish to acknowledge the major contributions made by Peter Markstein of the Hewlett-Packard Company to the design and further improvement of many algorithms described in [1].

8. References

[1] *Division, Square Root, and Remainder Algorithms for the Intel® Itanium™ Architecture, Application Note*, Document Nr. 248725-004, Intel Corporation, 2003, http://developer.intel.com/software/products/opensource/libraries/num.htm

[2] *ANSI/IEEE Std 754-1985, IEEE Standard for Binary Floating-Point Arithmetic*, IEEE, New York, 1985.

[3] Intel Corporation, *Intel® Itanium™ Architecture Software Developer's Manual*, http://developer.intel.com/design/itanium/manuals.htm

[4] Markstein, P., *Computation of Elementary Functions on the IBM RISC System/6000 Processor*, IBM Journal, 1990

[5] Cornea-Hasegan, M., *Proving IEEE Correctness of Iterative Floating-Point Square Root, Divide, and Remainder Algorithms,* Intel Technology Journal, Q2 , 1998, http://developer.intel.com/technology/itj/q21998.htm

[6] Cornea-Hasegan, M. and Golliver, R., Markstein, P., *Correctness Proofs Outline for Newton- Raphson Based Floating-Point Divide and Square Root Algorithms,* Proceedings of the 14th IEEE Symposium on Computer Arithmetic, 1999, IEEE Computer Society, Los Alamitos, CA, pp. 96-105.

[7] Cornea-Hasegan, M. and Norin, B., *IA-64 Floating-Point Operations and the IEEE Standard for Binary Floating-Point Arithmetic,* Intel Technology Journal, Q4, 1999, http://developer.intel.com/technology/itj/q41999.htm

[8] Cornea-Hasegan, M., Iordache, C., Harrison, J. and Markstein, P., *Integer Divide and Remainder Operations in the IA-64 Architecture,* Proceedings of the 4th Conference on Real Numbers and Computers, Germany, April 2000.

[9] Intel Corporation, *Intel® Itanium™ Processor Floating-Point Software Assistance Handler*, http://developer.intel.com/software/products/opensource/libraries/num.htm

[10] Marius Cornea, John Harrison, Ping Tak Peter Tang, *Scientific Computing on Itanium-based Systems,* Intel Press, 2002